

Polynomial self-stabilizing algorithm and proof for a 2/3-approximation of a maximum matching

Johanne Cohen¹, Khaled Maâmra², George Manoussakis¹, and Laurence Pilard²

¹LRI-CNRS, Université Paris-Sud, Université Paris Saclay, France,
 {johanne.cohen, george.manoussakis}@lri.fr

²LI-PaRAD, Université Versailles-St. Quentin, Université Paris Saclay, France,
 {khaled.maamra, laurence.pilard}@uvsq.fr

Abstract

We present the first polynomial self-stabilizing algorithm for finding a $\frac{2}{3}$ -approximation of a maximum matching in a general graph. The previous best known algorithm has been presented by Manne *et al.* [6] and has a sub-exponential time complexity under the distributed adversarial daemon [1]. Our new algorithm is an adaptation of the Manne *et al.* algorithm and works under the same daemon, but with a time complexity in $O(n^3)$ moves. Moreover, our algorithm only needs one more boolean variable than the previous one, thus as in the Manne *et al.* algorithm, it only requires a constant amount of memory space (three identifiers and *two* booleans per node).

1 Introduction

In graph theory, a *matching* M in a graph G is a subset of the edges of G without common nodes. A matching is *maximal* if no proper superset of M is also a matching whereas a *maximum* matching is a maximal matching with the highest cardinality among all possible maximal matchings. Some (almost) linear time approximation algorithm for the maximum weighted matching problem have been well studied [3, 7], nevertheless these algorithms are not distributed. They are based on a simple greedy strategy using *augmenting path*. An *augmenting path* is a path, starting and ending in an unmatched node, and where every other edge is either unmatched or matched; *i.e.* for each consecutive pair of edges, exactly one of them must belong to the matching. Let us consider the example in Figure 1d, page 5. In this figure, u and v are matched nodes and x, y are unmatched nodes. The path (x, u, v, y) is an augmenting path of length 3 (written *3-augmenting path*). It is well known [4] that given a graph $G = (V, E)$ and a matching $M \subseteq E$, if there is no augmenting path of length $2k - 1$ or less, then M is a $\frac{k}{k+1}$ -approximation of the maximum matching. See [3] for the weighted version of this theorem. The greedy strategy in [3, 7] consists in finding all augmenting paths of length ℓ or less and by switching matched and unmatched edges of these paths in order to improve the maximum matching approximation.

In this paper, we present a self-stabilizing algorithm for finding a maximum matching with approximation ratio $2/3$ that uses the greedy strategy presented above. Our algorithm stabilizes after $O(n^3)$ moves under the adversarial distributed daemon.

2 Model

The system consists of a set of processes where two adjacent processes can communicate with each other. The communication relation is represented by an undirected graph $G = (V, E)$ where $|V| = n$ and $|E| = m$. Each process corresponds to a node in V and two processes u and v are adjacent if and only if $(u, v) \in E$. The set of neighbors of a process u is denoted by $N(u)$ and is the set of all processes adjacent to u , and Δ is the maximum degree of G . We assume all nodes in the system have a unique identifier.

For the communication, we consider the *shared memory model*. In this model, each process maintains a set of *local variables* that makes up the *local state* of the process. A process can read its local variables

and the local variables of its neighbors, but it can write only in its own local variables. A *configuration* C is the local states of all processes in the system. Each process executes the same algorithm that consists of a set of *rules*. Each rule is of the form of $\langle \text{name} \rangle :: \text{if } \langle \text{guard} \rangle \text{ then } \langle \text{command} \rangle$. The *name* is the name of the rule. The *guard* is a predicate over the variables of both the process and its neighbors. The *command* is a sequence of actions assigning new values to the local variables of the process.

A rule is *activable* in a configuration C if its guard in C is true. A process is *eligible* for the rule \mathcal{R} in a configuration C if its rule \mathcal{R} is activable in C and we say the process is *activable* in C . An *execution* is an alternate sequence of configurations and actions $\mathcal{E} = C_0, A_0, \dots, C_i, A_i, \dots$, such that $\forall i \in \mathbb{N}^*$, C_{i+1} is obtained by executing the command of at least one rule that is activable in C_i (a process that executes such a rule makes a *move*). More precisely, A_i is the non empty set of activable rules in C_i that has been executed to reach C_{i+1} and such that each process has at most one of its rules in A_i . We use the notation $C_i \mapsto C_{i+1}$ to denote this transition in \mathcal{E} . Finally, let $\mathcal{E}' = C'_0, A'_0, \dots, C'_k$ be a finite execution. We say \mathcal{E}' is a *sub-execution* of \mathcal{E} if and only if $\exists t \geq 0$ such that $\forall j \in [0, \dots, k]: (C'_j = C_{j+t} \wedge A'_j = A_{j+t})$.

An *atomic operation* is such that no change can take place during its run, we usually assume that an atomic operation is instantaneous. In the shared memory model, a process u can read the local state of all its neighbors and update its whole local state in one atomic step. Then, we assume here that a rule is an atomic operation. An execution is *maximal* if it is infinite, or it is finite and no process is activable in the last configuration. All algorithm executions considered here are assumed to be maximal.

A *daemon* is a predicate on the executions. We consider only the most powerful one: the *adversarial distributed daemon* that allows all executions described in the previous paragraph. Observe that we do not make any fairness assumption on the executions.

An algorithm is *self-stabilizing* for a given specification, if there exists a sub-set \mathcal{L} of the set of all configurations such that: every execution starting from a configuration of \mathcal{L} verifies the specification (*correctness*) and starting from any configuration, every execution eventually reaches a configuration of \mathcal{L} (*convergence*). \mathcal{L} is called the set of *legitimate configurations*. A configuration is *stable* if no process is activable in the configuration. The algorithm presented here, is *silent*, meaning that once the algorithm has stabilized, no process is activable. In other words, all executions of a silent algorithm are finite and end in a stable configuration. Note the difference with a non silent self-stabilizing algorithm that has at least one infinite execution with a suffix only containing legitimate configurations, but not stable ones.

3 Algorithm

The algorithm presented in this paper is called MAXMATCH, and is based on the algorithm presented by Manne *et al.* [6]. As in the Manne *et al.* algorithm, MAXMATCH assumes there exists an underlying maximal matching algorithm, which has reached a stable configuration. Based on this stable maximal matching, MAXMATCH builds a $\frac{2}{3}$ -approximation of the maximum matching by detecting and then deleting all 3-augmenting paths. Once a 3-augmenting path is detected, nodes rearrange the matching accordingly, *i.e.*, transform this path with one matched edge into a path with two matched edges. This transformation leads to the deletion of the augmenting path and increases by one the cardinality of the matching. The algorithm stabilizes when there is no augmenting path of length three left. By the result of Hopcroft *et al.* [4], we obtain a $\frac{2}{3}$ -approximation of the maximum matching.

This underlying stabilized maximal matching can be built, for instance, with the self-stabilizing maximal matching algorithm from [5] that stabilizes in $O(m)$ moves under the adversarial distributed daemon (so the same daemon than the one used in this paper). Observe that this algorithm is silent, meaning that the maximal matching remains constant once the algorithm has stabilized. Then, using a classical composition of this two algorithms [2], we obtain a total time complexity in $O(n^2 \times n^3) = O(n^5)$ moves under the adversarial distributed daemon.

In the rest of the paper, \mathcal{M} is the underlying maximal matching, and \mathcal{M}^+ is the set of edges built by our algorithm MAXMATCH (see Definition 1). For a set of nodes A , we define *single*(A) and *matched*(A) as the set of unmatched and matched nodes in A , accordingly to the underlying maximal matching \mathcal{M} . Moreover, \mathcal{M} is encoded with the variable m_u . If $(u, v) \in \mathcal{M}$ then u and v are *matched nodes* and we have: $m_u = v \wedge m_v = u$. If u is not incident to any edge in \mathcal{M} , then u is a *single node* and $m_u = \text{null}$. Since we assume the underlying maximal matching is stable, a node membership in *matched*(V) or *single*(V) will not change, and each node u can use the value of m_u to determine which set it belongs to.

Variables description: In order to facilitate the rematching, each node $u \in V$ maintains three pointers and two boolean variables. The pointer p_u refers to a neighbor of u that u is trying to (re)match with. If $p_u = \text{null}$ then the matching of u has not changed from the maximal matching. Thus, the matching \mathcal{M}^+ built by our algorithm is defined as follows:

Definition 1. *The set of edges built by algorithm MAXMATCH is $\mathcal{M}^+ = \{(u, v) \in \mathcal{M} : p_u = p_v = \text{null}\} \cup \{(a, b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$*

For a matched node u , pointers α_u and β_u refer to two nodes in $\text{single}(N(u))$ that are *candidates* for a possible rematching with u . Also, s_u is a boolean variable that indicates if the node u has performed a successful rematching with its single node candidate. Finally, end_u is a boolean variable that indicates if both u and m_u have performed a successful rematching or not. For a single node x , only one pointer p_x and one boolean variable end_x are needed. p_x has the same purpose as the p -variable of a matched node. The end -variable of a single node allows the matched nodes to know whether it is *available* or not. A single node is *available* if it is possible for this node to eventually rematch with one of its neighboring married node, i.e., $\text{end}_x = \text{False}$.

In our algorithm, $\text{Unique}(A)$ returns the number of unique elements in the multi-set A , and $\text{Lowest}(A)$ returns the node in A with the lowest identifier. If $A = \emptyset$, then $\text{Lowest}(A)$ returns null . Moreover, rules have priorities. In the algorithm, we present rules from the highest priority (at the top) to the lowest one (at the bottom).

Graphical convention: We will follow the above conventions in all the figures: matched nodes are represented with double circles and single nodes with simple circles. Moreover, all edges that belong to the maximal matching \mathcal{M} are represented with a double line, whereas the other edges are represented with a simple line. Black arrows show the content of the local variable p . If the p -value is null , we draw a 'T'. A prohibited value is first drawn in grey, then scratched out in black. If there is no knowledge on the p -value, nothing is drawn. For instance, in Figure 1e, page 5, x is a single node, u and v are matched nodes and $(u, v) \in \mathcal{M}$, $p_u = x$, and $p_x \neq u$. In Figure 1d page 5, $p_u = \perp$.

Now, we present the proof of our algorithm.

4 Correctness Proof

We first introduce some notations. A matched node u is said to be *First* if $\text{AskFirst}(u) \neq \text{null}$. In the same way, u is *Second* if $\text{AskSecond}(u) \neq \text{null}$. Let $\text{Ask} : V \rightarrow V \cup \{\text{null}\}$ be a function where $\text{Ask}(u) = \text{AskFirst}(u)$ if $\text{AskFirst}(u) \neq \text{null}$, otherwise $\text{Ask}(u) = \text{AskSecond}(u)$. We will say a node makes a *match* rule if it performs a *MatchFirst* or *MatchSecond* rule.

If C and C' are two configurations in \mathcal{E} , then we note $C \leq C'$ if and only if C appears before C' in \mathcal{E} or if $C = C'$. Moreover, we write $\mathcal{E} \setminus C$ to denote all configurations of \mathcal{E} except configuration C .

Definition 2. *Let $G = (V, E)$ be a graph and M be a maximal matching of G . (x, u, v, y) is a 3-augmenting path on (G, M) if:*

1. (x, u, v, y) is a path in G (so all nodes are distincts);
2. $\{(x, u), (v, y)\} \subset E \setminus M$;
3. $(u, v) \in M$

For instance, in Figure 1d, (x, u, v, y) is a 3-augmenting path.

Recall that the set of edges built by our algorithm MAXMATCH is $\mathcal{M}^+ = \{(u, v) \in \mathcal{M} : p_u = p_v = \text{null}\} \cup \{(a, b) \in E \setminus \mathcal{M} : p_a = b \wedge p_b = a\}$.

For the correctness part of the proof, we prove that in a stable configuration, \mathcal{M}^+ is a 2/3-approximation of a maximum matching on graph G . To do that we demonstrate there is no 3-augmenting path on (G, \mathcal{M}^+) . In particular we prove that for any edge $(u, v) \in \mathcal{M}$, we have either $p_u = p_v = \text{null}$, or u and v have two distincts single neighbors they are rematched with, i.e., $\exists x \in \text{single}(N(u)), \exists y \in \text{single}(N(v))$

Rules for nodes in single(V)

SingleNode - ResetEnd -- Priority 1 (the strongest) --

if $p_u = \text{null} \wedge \text{end}_u = \text{True}$
then $\text{end}_u := \text{False}$

SingleNode - UpdateP -- Priority 2 --

if $(p_u = \text{null} \wedge \{w \in \text{matched}(N(u)) \mid p_w = u\} \neq \emptyset) \vee$
 $(p_u \notin (\text{matched}(N(u)) \cup \{\text{null}\})) \vee (p_u \neq \text{null} \wedge p_{p_u} \neq u)$
then $p_u := \text{Lowest}\{w \in N(u) \mid p_w = u\}$
 $\text{end}_u := \text{False}$

SingleNode - UpdateEnd -- Priority 3 --

if $(p_u \in \text{matched}(N(u)) \wedge p_{p_u} = u \wedge \text{end}_u \neq \text{end}_{p_u})$
then $\text{end}_u := \text{end}_{p_u}$

Rules for nodes in matched(V)

Update -- Priority 1 (the strongest) --

if $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin (\text{single}(N(u)) \cup \{\text{null}\})) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq \text{null}) \vee p_u \notin (\text{single}(N(u)) \cup \{\text{null}\}) \vee$
 $((\alpha_u, \beta_u) \neq \text{BestRematch}(u) \wedge (p_u = \text{null} \vee (p_{p_u} \neq u \wedge \text{end}_{p_u} = \text{True})))$
then $(\alpha_u, \beta_u) := \text{BestRematch}(u)$
 $(p_u, s_u, \text{end}_u) := (\text{null}, \text{False}, \text{False})$

MatchFirst -- Priority 2 --

if $(\text{AskFirst}(u) \neq \text{null}) \wedge$
 $[p_u \neq \text{AskFirst}(u) \vee$
 $s_u \neq (p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge p_{m_u} \in \{\text{AskSecond}(m_u), \text{null}\}) \vee$
 $\text{end}_u \neq (p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = \text{AskSecond}(m_u) \wedge \text{end}_{m_u})]$
then $\text{end}_u := (p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = \text{AskSecond}(m_u) \wedge \text{end}_{m_u})$
 $s_u := (p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge (p_{m_u} \in \{\text{AskSecond}(m_u), \text{null}\}))$
 $p_u := \text{AskFirst}(u)$

MatchSecond -- Priority 3 --

if $(\text{AskSecond}(u) \neq \text{null}) \wedge (s_{m_u} = \text{True}) \wedge$
 $[p_u \neq \text{AskSecond}(u) \vee \text{end}_u \neq (p_u = \text{AskSecond}(u) \wedge p_{p_u} = u \wedge p_{m_u} = \text{AskFirst}(m_u)) \vee s_u \neq \text{end}_u]$
then $\text{end}_u := (p_u = \text{AskSecond}(u) \wedge p_{p_u} = u \wedge p_{m_u} = \text{AskFirst}(m_u))$
 $s_u := \text{end}_u$
 $p_u := \text{AskSecond}(u)$

ResetMatch -- Priority 4 --

if $[(\text{AskFirst}(u) = \text{AskSecond}(u) = \text{null}) \wedge ((p_u, s_u, \text{end}_u) \neq (\text{null}, \text{False}, \text{False}))] \vee$
 $[\text{AskSecond}(u) \neq \text{null} \wedge p_u \neq \text{null} \wedge s_{m_u} = \text{False}]$
then $(p_u, s_u, \text{end}_u) := (\text{null}, \text{False}, \text{False})$

Predicates and functions

BestRematch(u)

$a = \text{Lowest } \{x \in \text{single}(N(u)) \mid (p_x = u \vee \text{end}_x = \text{False})\}$
 $b = \text{Lowest } \{x \in \text{single}(N(u)) \setminus \{a\} \mid (p_x = u \vee \text{end}_x = \text{False})\}$
return (a, b)

AskFirst(u)

if $\alpha_u \neq \text{null} \wedge \alpha_{m_u} \neq \text{null} \wedge 2 \leq \text{Unique}(\{\alpha_u, \beta_u, \alpha_{m_u}, \beta_{m_u}\})$
then if $\alpha_u < \alpha_{m_u} \vee (\alpha_u = \alpha_{m_u} \wedge \beta_u = \text{null}) \vee (\alpha_u = \alpha_{m_u} \wedge \beta_{m_u} \neq \text{null} \wedge u < m_u)$
then return α_u
else return null

AskSecond(u)

if $\text{AskFirst}(m_u) \neq \text{null}$
then return $\text{Lowest}(\{\alpha_u, \beta_u\} \setminus \{\alpha_{m_u}\})$
else return null

with $x \neq y$ such that $(p_x = u) \wedge (p_u = x) \wedge (p_y = v) \wedge (p_v = y)$. In order to prove that, we show every other case for (u, v) is impossible. Main studied cases are shown in Figure 1. Finally, we prove that if $p_u = p_v = \text{null}$ then (u, v) does not belong to a 3-augmenting-path on (G, \mathcal{M}^+) .

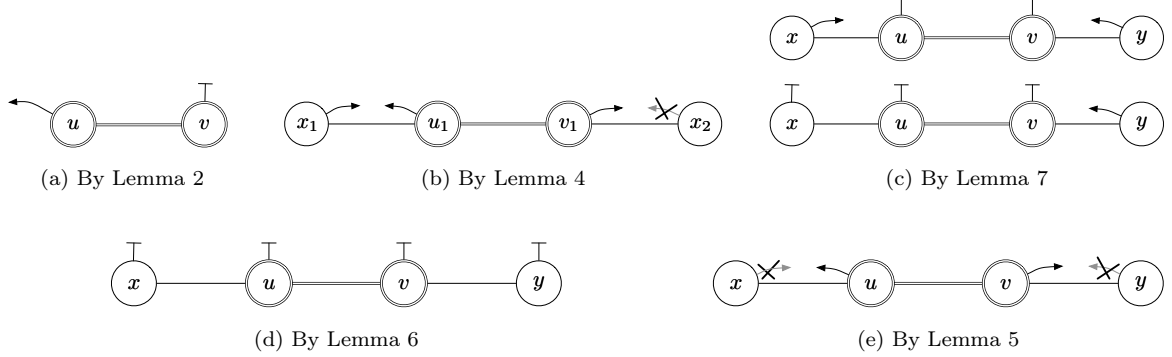


Figure 1: Impossible situations in a stable configuration.

Lemma 1. *In any stable configuration, we have the following properties:*

- $\forall u \in \text{matched}(V) : p_u = \text{Ask}(u)$;
- $\forall x \in \text{single}(V) : \text{if } p_x = u \text{ with } u \neq \text{null}, \text{ then } u \in \text{matched}(N(x)) \wedge p_u = x \wedge \text{end}_u = \text{end}_x$.

Proof. First, we will prove the first property. We consider the case where $\text{AskFirst}(u) \neq \text{null}$. We have $p_u = \text{AskFirst}(u)$, otherwise node u can execute rule AskFirst . We can apply the same result for the case where $\text{AskSecond}(u) \neq \text{null}$. Finally, we consider the case where $\text{AskFirst}(u) = \text{AskSecond}(u) = \text{null}$. If $p_u \neq \text{null}$, then node u can execute rule ResetMatch which yields the contradiction. Thus, $p_u = \text{null}$.

Second, we consider a stable configuration C where $p_x = u$, with $u \neq \text{null}$. $u \in \text{matched}(N(x))$, otherwise x is eligible for an UpdateP rule. Now there are two cases: $p_u = x$ and $p_u \neq x$. If $p_u \neq x$, this means that $p_{p_x} \neq x$. Thus, x is eligible for rule UpdateP , and this yields to a contradiction with the fact that C is stable. Finally, we have $\text{end}_u = \text{end}_x$, otherwise x is eligible for rule UpdateEnd . \square

Lemma 2. *Let (u, v) be an edge in \mathcal{M} . Let C be a configuration. If $p_u \neq \text{null} \wedge p_v = \text{null}$ holds in C (see Figure 1a), then C is not stable.*

Proof. By contraction. We assume C is stable. From Lemma 1, we have $p_u = \text{Ask}(u) \neq \text{null}$ and $p_v = \text{Ask}(v)$. So, by definition of predicates AskFirst and AskSecond , $\text{Ask}(u) = x \neq \text{null}$ implies that $\text{Ask}(v) \neq \text{null}$. This contradicts that fact that $p_v = \text{Ask}(v) = \text{null}$. \square

Lemma 3. *Let (x, u, v, y) be a 3-augmenting path on (G, \mathcal{M}) . Let C be a stable configuration. In C , if $p_x = u$, $p_u = x$, $p_v = y$ and $p_y = u$, then $\text{end}_x = \text{end}_u = \text{end}_v = \text{end}_y = \text{True}$.*

Proof. From Lemma 1, $p_u = \text{Ask}(u)$ (resp. $p_v = \text{Ask}(v)$) thus $\text{Ask}(u) \neq \text{null}$ and $\text{Ask}(v) \neq \text{null}$. W.l.o.g, we can assume that $\text{AskFirst}(u) \neq \text{null}$. We have $s_u = \text{True}$, otherwise u can execute MatchFirst rule. Now, as $s_u = \text{True}$, we must have $\text{end}_v = \text{True}$, otherwise v can execute MatchSecond rule. As $s_u = \text{end}_v = \text{True}$, we must have $\text{end}_u = \text{True}$, otherwise u can execute MatchFirst rule. From Lemma 1, we can deduce that $\text{end}_x = \text{end}_u = \text{end}_v = \text{end}_y = \text{True}$ and this concludes the proof. \square

Lemma 4. *Let (x_1, u_1, v_1, x_2) be a 3-augmenting path on (G, \mathcal{M}) . Let C be a configuration. If $p_{x_1} = u_1 \wedge p_{u_1} = x_1 \wedge p_{v_1} = x_2 \wedge p_{x_2} \neq v_1$ holds in C (see Figure 1b), then C is not stable.*

Proof. By contraction. We assume C is stable. From Lemma 1, $\text{Ask}(u_1) = x_1$ and $\text{Ask}(v_1) = x_2$.

First we assume that $\text{AskSecond}(u_1) = x_1$ and $\text{AskFirst}(v_1) = x_2$. The local variable s_{v_1} is *False*, otherwise v_1 would be eligible for executing the MatchFirst rule. Since $\text{AskSecond}(u_1) \neq \text{null} \wedge p_{u_1} \neq \text{null} \wedge s_{v_1} = \text{False}$, this implies that u_1 is eligible for the ResetMatch rule which is a contradiction.

Second, we assume that $AskFirst(u_1) = x_1$ and $AskSecond(v_1) = x_2$. We have $s_{u_1} = True$, otherwise u_1 can execute the *MatchFirst* rule. This implies that $end_{v_1} = False$, otherwise v_1 can execute the *MatchSecond* rule. As $end_{v_1} = False$, then $end_{u_1} = False$, otherwise u_1 can execute the *MatchFirst* rule. From Lemma 1, $end_{x_1} = end_{u_1} = end_{v_1} = False$. Since $Ask(v_1) = x_2$, we have $x_2 \in \{\alpha_{v_1}, \beta_{v_1}\}$. Let us assume $end_{x_2} = True$. Then $x_2 \notin BestRematch(v_1)$ and then v_1 is eligible for an *Update*. Thus $end_{x_2} = False$.

Therefore, C is a configuration such that u_1 is *First* and v_1 is *Second* with $end_{x_1} = end_{u_1} = end_{v_1} = end_{x_2} = False$. Now we are going to show there exists another augmenting path (x_2, u_2, v_2, x_3) with $end_{x_2} = end_{u_2} = end_{v_2} = end_{x_3} = False$ and $p_{u_2} = x_2$, $p_{x_2} = u_2$, $p_{v_2} = x_3$ and $p_{x_3} \neq v_2$ such that u_2 is *First* and v_2 is *Second* (see Figure 2).

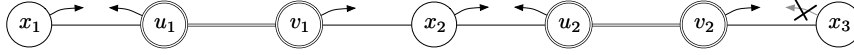


Figure 2: A chain of 3-augmenting paths.

$p_{x_2} \neq null$ otherwise x_2 is eligible for an *UpdateP* rule. Thus there exists a vertex $u_2 \neq v_1$ such that $p_{x_2} = u_2$. From Lemma 1, $u_2 \in matched(N(x_2))$ and $p_{u_2} = x_2$. Therefore, there exists a node $v_2 = m_{u_2}$. From Lemma 2, we can deduce that $p_{v_2} \neq null$ and there exists a node x_3 such that $p_{v_2} = x_3$. $x_3 \in single(N(v_2))$ otherwise x_2 is eligible for an *Update* rule. Finally, if $p_{x_3} = v_2$, then Lemma 3 implies that $end_{x_2} = end_{u_2} = end_{v_2} = end_{x_3} = True$. This yields to the contradiction with the fact $end_{x_2} = False$. So, we have $p_{x_3} \neq v_2$.

We can then conclude that (x_2, u_2, v_2, x_3) is a 3-augmenting path such that $p_{x_2} = u_2 \wedge p_{u_2} = x_2 \wedge p_{v_2} = x_3 \wedge p_{x_3} \neq v_2$. This augmenting path has the exact same properties than the first considered augmenting path (x_1, u_1, v_1, x_2) and in particular u_1 is *First*.

Now we can continue the construction in the same way. Therefore, for C to be stable, it has to exist a chain of 3-augmenting paths $(x_1, u_1, v_1, x_2, u_2, v_2, x_3, \dots, x_i, u_i, v_i, x_{i+1}, \dots)$ where $\forall i \geq 1 : (x_i, u_i, v_i, x_{i+1})$ is a 3-augmenting path with $p_{x_i} = u_i \wedge p_{u_i} = x_i \wedge p_{v_i} = x_{i+1} \wedge p_{x_{i+1}} = v_{i+1}$ and u_i is *First*. Thus, $x_1 < x_2 < \dots < x_i < \dots$ since the u_i will always be *First*. Since the graph is finite some x_k must be equal to some x_ℓ with $\ell \neq k$ which contradicts the fact that the identifier' sequence is strictly increasing. \square

Lemma 5. Let (x, u, v, y) be a 3-augmenting path on (G, \mathcal{M}) . Let C be a configuration. If $p_u = x \wedge p_x \neq u \wedge p_v = y \wedge p_y \neq v$ holds in C (see Figure 1e), then C is not stable.

Proof. By contradiction, assume C is stable. From Lemma 1, $Ask(u) = x$. Assume to begin that $AskFirst(u) \neq null$. Because $p_{p_u} \neq u$ we have $s_u = False$, otherwise u is eligible for *MatchFirst*. Since $AskSecond(v) \neq null$ and $s_{m_v} = s_u = False$ then v can apply the *ResetMatch* rule which yields a contradiction. Therefore assume that $AskSecond(u) \neq null$. The situation is symmetric (because now $AskFirst(v) \neq null$) and therefore we get the same contradiction as before. \square

Lemma 6. Let (x, u, v, y) be a 3-augmenting path on (G, \mathcal{M}) . Let C be a configuration. If $p_y = p_u = p_v = null$ holds in C (see Figure 1d), then C is not stable.

Proof. By contradiction, assume C is stable. $end_x = False$ (resp. $end_y = False$), otherwise x (resp. y) is eligible for a *ResetMatch*. $(\alpha_u, \beta_u) = BestRematch(u)$ (resp. $(\alpha_v, \beta_v) = BestRematch(v)$), otherwise u (resp. v) is eligible for an *Update*. Thus, there is at least an available single node for u and v and so $Ask(u) \neq null$ and $Ask(v) \neq null$. Then, this contradicts the fact that $Ask(u) = null$ (see Lemma 1). \square

Theorem 1. In a stable configuration we have, $\forall (u, v) \in \mathcal{M}$:

- $p_u = p_v = null$ or
- $\exists x \in single(N(u)), \exists y \in single(N(v))$ with $x \neq y$ such that $p_x = u \wedge p_u = x \wedge p_y = v \wedge p_v = y$.

Proof. We will prove that all cases but these two are not possible in a stable configuration. First, Lemma 2 says the configuration cannot be stable if exactly one of p_u or p_v is not *null*.

Second, assume that $p_u \neq null \wedge p_v \neq null$. Let $p_u = x$ and $p_v = y$. Observe that $x \in single(N(u))$ (resp. $y \in single(N(v))$), otherwise u (resp. v) is eligible for *Update*.

Case $x \neq y$: If $p_x \neq u$ and $p_y \neq v$ then Lemma 5 says the configuration cannot be stable. If $p_x = u$ and $p_y \neq v$ then Lemma 4 says the configuration cannot be stable. Thus, the only remaining possibility when $p_u \neq \text{null}$ and $p_v \neq \text{null}$ is: $p_x = u$ and $p_y = v$.

Case $x = y$: $\text{Ask}(u)$ (resp. $\text{Ask}(v) \neq \text{null}$), otherwise u (resp. v) is eligible for a *ResetMatch*. W.l.o.g. let us assume that u is First. $x = \text{AskFirst}(u)$ (resp. $x = \text{AskSecond}(v)$), otherwise u (resp. v) is eligible for *MatchFirst* (resp. *MatchSecond*). Thus $\text{AskFirst}(u) = \text{AskSecond}(v)$ which is impossible according to these two predicates. \square

Lemma 7. *Let x be a single node. In a stable configuration, if $p_x = u, u \neq \text{null}$ then it exists a 3-augmenting path (x, u, v, y) on (G, \mathcal{M}) such that $p_x = u \wedge p_u = x \wedge p_v = y \wedge p_y = v$.*

Proof. By lemma 1, if $p_x = u$ with $u \neq \text{null}$ then $u \in \text{matched}(N(x))$ and $p_u = x$. Since $p_u \neq \text{null}$, by Theorem 1 the result holds. \square

Observe that according to this Lemma, cases from Figure 1c are impossible.

Thus, in a stable configuration, for all edges $(u, v) \in \mathcal{M}$, if $p_u = p_v = \text{null}$ then (u, v) does not belong to a 3-augmenting-path on (G, \mathcal{M}^+) . In other words, we obtain:

Corollary 1. *In a stable configuration, there is no 3-augmenting path on (G, \mathcal{M}^+) left.*

5 Convergence Proof

We start by giving a sketch of the convergence proof. In the following, μ denotes the number of matched nodes and σ the number of single nodes.

First, we focus on the variable *end*. Recall that for a matched edge (u, v) , the variable *end* indicates if both u and v have performed a successful rematch or not. In section 5.1, we prove the maximum number of times a matched node u can write *True* in its variable *end* is 2. The idea is that only one writing can correspond to an incorrect initialization of the node m_u .

Theorem 2. *In any execution, a matched node u can write $\text{end}_u := \text{True}$ at most twice.*

This theorem is the key point of the convergence proof. Indeed, this result exhibits an action that can be performed only a finite number of time. Thus, with this result, we have a first strong step leading to the proof of the silent property of our algorithm.

After considering the *end* variable of matched nodes in section 5.1, we focus on the *end* variable of single nodes in section 5.2. Since single nodes just follow orders from their neighboring matched nodes (Lemma 17), we can count the number of times single nodes can change the value of their *end* variable as described above. There are σ possible modifications due to bad initializations. Moreover, a matched node u can write *True* twice in end_u , so end_u can be *True* during 3 distinct sub-executions. As a single node x copies the *end*-value of the matched node it points to ($p_x = u$), then a single node can write *True* in its *end*-variable at most 3 times as well. So we obtain 6μ modifications.

Lemma 19. *In any execution, the number of transitions where a single node changes the value of its end variables (from *True* to *False* or from *False* to *True*) is at most $\sigma + 6\mu$ times.*

In section 5.3, we count the maximal number of *Update* rule that can be performed in any execution. To do that, we observe that the first line of the *Update* guard can be *True* at most once in an execution (Lemma 10). Then we prove for the second line of the guard to be *True*, a single node has to change its *end* value (Lemma 20). Thus, for each single node modification of the *end*-value, at most all matched neighbors of this single node can perform one *Update* rule.

Corollary 3. *Matched nodes can execute at most $\Delta(\sigma + 6\mu) + \mu$ times the *Update* rule.*

In section 5.4, we count the maximal number of moves performed by matched nodes between two *Update*. The idea is that in an execution without *Update*, α and β values of all matched nodes remain constant. Thus, in these small executions, at most one augmenting path is detected per matched edge and at most one rematch attempt is performed per matched edge. We obtain that the maximal number of moves of a matched node in these small executions is 12 (Lemma 26). By the previous remark and Corollary 3, we obtain:

Theorem 3. *In any execution, matched nodes can execute at most $12\mu(\Delta(\sigma + 6\mu) + \mu)$ rules.*

Finally, we count the maximal number of moves that single nodes can perform, counting rule by rule. The *ResetEnd* is done at most once (Lemma 27). The number of *UpdateEnd* is bounded by the number of times single nodes can change their *end*-value, so it is at most $\sigma + 6\mu$ (Lemma 28). Finally, *UpdateP* is counted as follows: between two consecutive *UpdateP* executed by a single node x , a matched node has to make a move. The total number of executed *UpdateP* is then at most $12\mu(\Delta(\sigma + 6\mu) + \mu) + 1$ (See Lemma 29).

Corollary 5. *The algorithm MAXMATCH converges in $O(n^3)$ steps under the adversarial distributed daemon.*

5.1 A matched node can write *True* in its *end*-variable at most twice

The first three lemmas are technical lemmas.

Lemma 8. *Let u be a matched node. Consider an execution \mathcal{E} starting after u executed some rule. Let C be any configuration in \mathcal{E} . If $\text{end}_u = \text{True}$ in C then $s_u = \text{True}$ as well.*

Proof. Let $C_0 \mapsto C_1$ be the transition in \mathcal{E} in which u executed a rule for the last time before C . Observe that C may be equal to C_1 . The executed rule is necessarily a *match* rule, otherwise end_u could not be *True* in C_1 . If it is a *MatchSecond* the lemma holds since in that case s_u is a copy of end_u . Assume now it is a *MatchFirst*. For end_u to be *True* in C_1 , $p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge p_{m_u} = \text{AskSecond}(m_u)$ must hold in C_0 , according to the guard of *MatchFirst*. This implies that u writes *True* in s_u in transition $C_0 \mapsto C_1$. \square

Lemma 9. *Let u be a matched node. Consider an execution \mathcal{E} starting after u executed some rule. Let C be any configuration in \mathcal{E} . In C , if $s_u = \text{True}$ then $\exists x \in \text{single}(N(u)) : p_u = x \wedge p_x = u$.*

Proof. Consider transition $C_0 \mapsto C_1$ in which u executed a rule for the last time before C . The executed rule is necessarily a *match* rule, otherwise s_u could not be *True* in C_1 . Observe now that whichever *match* rule is applied, $\text{Ask}(u) \neq \text{null}$ – let us assume $\text{Ask}(u) = x$ – and $p_u = x$ and $p_x = u$ must hold in C_0 for s_u to be *True* in C_1 . $p_u = x$ still holds in C_1 and until C . Moreover, x must be in $\text{single}(N(u))$, otherwise u would have executed an *Update* instead of a *match* rule in $C_0 \mapsto C_1$, since *Update* has the highest priority among all rules. Finally, in transition $C_0 \mapsto C_1$, x cannot execute *UpdateP* nor *ResetEnd* since $p_x \in \text{matched}(N(x)) \wedge p_{p_x} = x$ holds in C_0 . Thus in C_1 , $p_u = x$ and $p_x = u$ holds. Using the same argument, x cannot execute *UpdateP* nor *ResetEnd* between configurations C_1 and C . Thus $p_u = x \wedge p_x = u$ in C . \square

Lemma 10. *Let u be a matched node and \mathcal{E} be an execution containing a transition $C_0 \mapsto C_1$ where u makes a move. From C_1 , the predicate in the first line of the guard of the *Update* rule will ever hold from C_1 .*

Proof. Let C_2 be any configuration in \mathcal{E} such that $C_2 \geq C_1$. Let $C_{10} \mapsto C_{11}$ be the last transition before C_2 in which u executes a move. Notice that by definition of \mathcal{E} , this transition exists. Assume by contradiction that one of the following predicates holds in C_2 .

1. $(\alpha_u > \beta_u) \vee (\alpha_u, \beta_u \notin (\text{single}(N(u)) \cup \{\text{null}\})) \vee (\alpha_u = \beta_u \wedge \alpha_u \neq \text{null})$
2. $p_u \notin (\text{single}(N(u)) \cup \{\text{null}\})$

By definition between C_{11} and C_2 , u does not execute rules. To modify the variables α_u, β_u and p_u , u must execute a rule. Thus one of the two predicates also holds in C_{11} .

We first show that if predicate (1) holds in C_{11} then we get a contradiction. If u executes an *Update* rule in transition $C_{10} \mapsto C_{11}$, then by definition of the *BestRematch* function, predicate (1) cannot hold in C_{11} (observe that the only way for $\alpha_u = \beta_u$ is when $\alpha_u = \beta_u = \text{null}$). Thus assume that u executes a *match* or *ResetMatch* rule. Notice that these rules do not modify the value of the α_u and β_u variables. This implies that if u executes one of these rules in $C_{10} \mapsto C_{11}$, predicate (1) not only hold in C_{11} but also in C_{10} . Observe that this implies, in that case that u is eligible for *Update* in $C_{10} \mapsto C_{11}$, which gives the contradiction since *Update* is the rule with the highest priority among all rules.

Now assume predicate (2) holds in C_{11} . In transition $C_{10} \mapsto C_{11}$, u cannot execute *Update* nor *ResetMatch* as this would imply that $p_u = \text{null}$ in C_{11} . Assume that in $C_{10} \mapsto C_{11}$ u executes a *match* rule. Since in C_{11} , $p_u \notin (\text{single}(N(u)) \cup \{\text{null}\})$ this implies that in C_{10} , $\text{Ask}(u) \notin (\text{single}(N(u)) \cup \{\text{null}\})$. This implies that $\alpha_u, \beta_u \notin (\text{single}(N(u)) \cup \{\text{null}\})$ in C_{10} . Thus u is eligible for *Update* in transition $C_{10} \mapsto C_{11}$ and this yields the contradiction since *Update* is the rule with the highest priority among all rules.

Since these two predicates cannot hold in C_2 , this concludes the proof. \square

Now, we focus on particular configurations for a matched edge (u, v) corresponding to the fact they have completely exploited a 3-augmenting path.

Lemma 11. *Let (u, v) be a matched edge, \mathcal{E} be an execution and C be a configuration of \mathcal{E} . If in C , we have:*

1. $p_u \in \text{single}(N(u)) \wedge p_u = \text{AskFirst}(u) \wedge p_{p_u} = u$;
2. $p_v \in \text{single}(N(v)) \wedge p_v = \text{AskSecond}(v) \wedge p_{p_v} = v$;
3. $s_u = \text{end}_u = s_v = \text{end}_v = \text{True}$;

then neither u nor v will ever be eligible for any rule from C .

Proof. Observe first that neither u nor v are eligible for any rule in C . Moreover, p_u (resp. p_v) is not eligible for an *UpdateP* move since u (resp. v) does not make any move. Thus p_{p_u} and p_{p_v} will remain constant since u and v do not make any move and so neither u nor v will ever be eligible for any rule from C . \square

The configuration C described in Lemma 11 is called a *stop_{uv}* configuration. From such a configuration neither u nor v will ever be eligible for any rule.

In Lemmas 13 and 14, we consider executions where a matched node u writes *True* in end_u twice, and we focus on the transition $C_0 \mapsto C_1$ where u performs its second writing. Lemma 13 shows that, if u is *First* in C_0 , then C_1 is a *stop_{um_u}* configuration. Lemma 14 shows that, if u is *Second* in C_0 , then either C_1 is a *stop_{um_u}* configuration or it exists a configuration C_3 such that $C_3 > C_1$, u does not make any move from C_1 to C_3 and C_3 is a *stop_{um_u}* configuration.

Lemma 12 and Corollary 2 are required to prove Lemmas 13 and 14.

Lemma 12. *Let (u, v) be a matched edge. Let \mathcal{E} be some execution in which v does not execute any rule. If it exists a transition $C_0 \mapsto C_1$ in \mathcal{E} where u writes *True* in end_u , then u is not eligible for any rule from C_1 .*

Proof. To write *True* in end_u in transition $C_0 \mapsto C_1$, u must have executed a *match* rule. According to this rule, $(p_u = \text{Ask}(u) \wedge p_{p_u} = u)$ holds C_0 with $p_u \in \text{single}(N(u))$, otherwise u would have executed an *Update* instead of a *match* rule. Now, in $C_0 \mapsto C_1$, p_u cannot execute *UpdateP* then it cannot change its p -value and v does not execute any move then it cannot change $\text{Ask}(u)$. Thus, $(p_u = \text{Ask}(u) \wedge p_{p_u} = u)$ holds in both C_0 and C_1 .

Assume now by contradiction that u executes a rule after configuration C_1 . Let $C_2 \mapsto C_3$ be the next transition in which it executes a rule. Recall that between configurations C_1 and C_2 both u and v do not execute rules. Observe also that p_u is not eligible for *UpdateP* between these configurations. Thus $(p_u = \text{Ask}(u) \wedge p_{p_u} = u)$ holds from C_0 to C_2 . Moreover the following points hold as well between C_0 and C_2 since in $C_0 \mapsto C_1$ u executed a *match* rule and v does not apply rules in \mathcal{E} :

- $\alpha_u, \alpha_v, \beta_u$ and β_v do not change.
- The values of the variables of v do not change.
- $\text{Ask}(u)$ and $\text{Ask}(v)$ do not change.
- If u was *First* in C_0 it is *First* in C_2 and the same holds if it was *Second*.

Using these remarks, we start by proving that u is not eligible for *ResetMatch* in C_2 . If it is *First* in C_2 , this holds since $\text{AskFirst}(u) \neq \text{null}$ and $\text{AskSecond}(u) = \text{null}$. If it is *Second* then to be eligible for *ResetMatch*, $s_v = \text{False}$ must hold in C_2 since $\text{AskSecond}(u) \neq \text{null}$. Since u executed $\text{end}_u = \text{True}$ in $C_0 \mapsto C_1$ and since u was *Second* in C_0 , then necessarily $s_v = \text{True}$ in C_0 and thus in C_2 (using remark 2 above). So u is not eligible for *ResetMatch* in C_2 .

We show now that u is not eligible for an *Update* in C_2 . The α and β variables of u and v remain constant between C_0 and C_2 . Thus if any of the three first disjunctions in the *Update* rule holds in C_2 then it also holds in C_0 and in $C_0 \mapsto C_1$ u should have executed an *Update* since it has higher priority than the *match* rules. Moreover since in C_2 $(p_u = \text{Ask}(u) \wedge p_{p_u} = u)$ holds, the last two disjunctions of *Update* are *False* and we can state u is not eligible for this rule.

We conclude the proof by showing that u is not eligible for a *match* rule in C_2 . If u was *First* in C_0 then it is *First* in C_2 . To write *True* in end_u then $(p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge s_u \wedge p_{m_u} = \text{AskSecond}(m_u) \wedge \text{end}_{m_u})$ must hold in C_0 . Since in $C_0 \mapsto C_1$ v does not execute rules, it also holds in C_1 . The same remark between configurations C_1 and C_2 implies that this predicate holds in C_2 . Thus in C_2 , all the three conditions of the *MatchFirst* guard are *False* and u not eligible for *MatchFirst*. A similar remark if u is *Second* implies that u will not be eligible for *MatchSecond* in C_2 if it was *Second* in C_0 . \square

Corollary 2. *Let (u, v) be a matched edge. In any execution, if u writes *True* in end_u twice, then v executes a rule between these two writing.*

Lemma 13. *Let (u, v) be a matched edge and \mathcal{E} be an execution where u writes *True* in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes *True* in end_u for the second time in \mathcal{E} . If u is *First* in C_0 then the following holds:*

1. in configuration C_0 ,
 - (a) $s_v = \text{end}_v = \text{True}$;
 - (b) $p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge s_u = \text{True} \wedge p_v = \text{AskSecond}(v)$;
 - (c) $p_u \in \text{single}(N(u))$;
 - (d) $p_v \in \text{single}(N(v)) \wedge p_{p_v} = v$;
2. v does not execute any move in $C_0 \mapsto C_1$;
3. in configuration C_1 ,
 - (a) $s_u = \text{end}_u = \text{True}$;
 - (b) $p_u \in \text{single}(N(u)) \wedge p_v \in \text{single}(N(v))$;
 - (c) $s_v = \text{end}_v = \text{True}$;
 - (d) $p_u = \text{AskFirst}(u) \wedge p_v = \text{AskSecond}(v)$;
 - (e) $p_{p_u} = u \wedge p_{p_v} = v$.

Proof. We prove Point 1a. Observe that for u to write *True* in end_u , end_v must be *True* in C_0 . By Lemma 8 this implies that s_v is *True* as well. Now Point 1b holds by definition of the *MatchFirst* rule. As in C_0 , u already executed an action, then according to Lemma 10, Point 1c holds and will always hold. By Corollary 2, u cannot write *True* consecutively if v does not execute moves. Thus at some point before C_0 , v applied some rule. This implies that in configuration C_0 , since $s_v = \text{True}$, by Lemma 9, $\exists x \in \text{single}(N(v)) : p_v = x \wedge p_x = v$. Thus Point 1d holds.

We now show that v does not execute any move in $C_0 \mapsto C_1$ (Point 2). Recall that v already executed an action before C_0 , so by Lemma 10, line 1 of the *Update* guard does not hold in C_0 . Moreover, by Point 1d, line 2 does not hold either. Thus, v is not eligible for *Update* in C_0 . We also have that $s_u = \text{True}$ and $\text{AskSecond}(v) \neq \text{null}$ in C_0 , thus v is not eligible for *ResetMatch*. Observe now that by Points 1a, 1b and 1d, v is not eligible for *MatchSecond* in C_0 . Finally v cannot execute *MatchFirst* since $\text{AskFirst}(v) = \text{null}$. Thus v does not execute any move in $C_0 \mapsto C_1$ and so Point 2 holds.

In C_1 , end_u is *True* by hypothesis and according to Point 1b, u writes *True* in s_u in transition $C_0 \mapsto C_1$. Thus Point 3a holds. Point 3b holds by Points 1c and 1d. Point 3c holds by Points 1a and 2. $\text{AskFirst}(u)$ and $\text{AskSecond}(v)$ remain constant in $C_0 \mapsto C_1$ since neither u nor v executes an *Update* in this transition. Moreover p_v remains constant in $C_0 \mapsto C_1$ by Point 2 and p_u remains constant also since it writes $\text{AskFirst}(u)$ in p_u in this transition while $p_u = \text{AskFirst}(u)$ in C_0 . Thus Point 3d holds. Observe that nor p_u neither p_v is eligible for an *UpdateP* in C_0 , thus Point 3e holds. \square

Now, we consider the case where u is *Second*.

Lemma 14. *Let (u, v) be a matched edge and \mathcal{E} be an execution where u writes *True* in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes *True* in end_u for the second time in \mathcal{E} . If u is *Second* in C_0 then the following holds:*

1. in configuration C_0 ,
 - (a) $s_v = \text{True} \wedge p_v = \text{AskFirst}(v)$;
 - (b) $p_v \in \text{single}(N(v)) \wedge p_{p_v} = v$;
2. in transition $C_0 \mapsto C_1$, v is not eligible for *Update* nor *ResetMatch*;
3. in configuration C_1 ,
 - (a) $s_u = \text{end}_u = \text{True}$;
 - (b) $p_v \in \text{single}(N(v)) \wedge p_v = \text{AskFirst}(v) \wedge p_{p_v} = v$;
 - (c) $p_u \in \text{single}(N(u)) \wedge p_u = \text{AskSecond}(u) \wedge p_{p_u} = u$;
 - (d) $s_v = \text{True}$;
4. u is not eligible for any move in C_1 ;
5. If $\text{end}_u = \text{False}$ in C_1 then the following holds:
 - (a) From C_1 , v executes a next move and this move is a *MatchFirst*;
 - (b) Let us assume this move (the first move of v from C_1) is done in transition $C_2 \mapsto C_3$. In configuration C_3 , we have:
 - i. $s_u = \text{end}_u = \text{True}$;
 - ii. $p_v \in \text{single}(N(v)) \wedge p_v = \text{AskFirst}(v) \wedge p_{p_v} = v$;
 - iii. $p_u \in \text{single}(N(u)) \wedge p_u = \text{AskSecond}(u) \wedge p_{p_u} = u$;
 - iv. $s_v = \text{True}$;
 - v. u does not execute moves between C_1 and C_3 ;
 - vi. $\text{end}_v = \text{True}$;

Proof. We show Point 1a. For u to write *True* in transition $C_0 \mapsto C_1$, u executes a *MatchSecond* in this transition. Thus $s_v = \text{True}$ must hold in C_0 and $p_v = \text{AskFirst}(v)$ as well. By Corollary 2, u cannot write *True* consecutively if v does not execute any move. Thus at some point before C_0 , v applied some rule. Thus, and by Lemma 9, $\exists x \in \text{single}(N(v)) : p_v = x \wedge p_x = v$ in configuration C_0 , so Point 1b holds.

As $\text{AskFirst}(v) \neq \text{null}$ in C_0 , v is not eligible for *ResetMatch* in C_0 . We prove now that v is not eligible for *Update*. By Corollary 2 and Lemma 10, line 1 of the *Update* guard does not hold in C_0 . Finally, according to Point 2b, the second line of the *Update* guard does not hold, which concludes Point 2.

We consider now Point 3a. In C_1 , $s_u = \text{end}_u = \text{True}$ holds because, executing a *MatchSecond*, u writes *True* in end_u and writes end_u in s_u during transition $C_0 \mapsto C_1$.

We now show Point 3b. $\text{AskFirst}(v)$ and $\text{AskSecond}(u)$ remain constant in $C_0 \mapsto C_1$ since neither u nor v execute an *Update* in this transition. Moreover, the only rule v can execute in $C_0 \mapsto C_1$ is a *MatchFirst*, according to Point 2. Thus v does not change its p -value in $C_0 \mapsto C_1$ and so $p_v = \text{AskFirst}(v)$ in C_1 . Now, in C_0 , $v \in \text{matched}(N(p_v)) \wedge p_{p_v} = v$ thus p_v cannot execute *UpdateP* in $C_0 \mapsto C_1$ and thus it cannot change its p -value. So, $p_{p_v} = v$ in C_1 .

Point 3c holds since after u executed a *MatchSecond* in $C_0 \mapsto C_1$, observe that necessarily $p_u = \text{AskSecond}(u)$ in C_1 . Moreover, $s_u = \text{True}$ in C_1 so, according to Lemma 9, $\exists y \in \text{single}(N(u)) : p_u = y \wedge p_y = u$ in C_1 .

$p_v = \text{AskFirst}(v)$ and $p_{p_v} = v$ hold in C_0 , according to Points 2a and 2b. Moreover, $p_u = \text{AskSecond}(u)$ holds in C_0 since u writes *True* in end_u while executing a *MatchSecond* in $C_0 \mapsto C_1$. Finally, by Point 2, v can only execute *MatchFirst* in $C_0 \mapsto C_1$, thus variable s_v remains *True* in transition $C_0 \mapsto C_1$ and Point 3d holds.

We now prove Point 4. If $\text{end}_v = \text{True}$ in C_1 , then according to Lemma 11, u is not eligible for any rule in C_1 . Now, let us consider the case $\text{end}_v = \text{False}$ in C_1 . By Points 3c and 3d, u is not eligible for *ResetMatch*. By Point 3c and Lemma 10, u is not eligible for *Update*. By Points 3a, 3b and 3c, u is not eligible for *MatchSecond*. Finally, since u is Second in C_1 , u is not eligible for *MatchFirst* neither and Point 4 holds.

Now since between C_1 and C_2 , v does not execute any rule (by Point 5b), and since p_u (resp. p_v) is not eligible for *UpdateP* while u (resp. v) does not move (because $p_{p_u} = u$ (resp. $p_{p_v} = v$)), then $\text{Ask}(u)$, $\text{Ask}(v)$, p_{p_u} and p_{p_v} remain constant while u does not make any move. And so, properties 3a, 3b, 3c and 3d hold for any configuration between C_1 and C_2 , thus u is not eligible for any rule between C_1 and C_2 and u will not execute any move from C_1 to C_3 . Moreover, the end_v -value is the same from C_1 to C_2 .

If $end_v = False$ in C_2 , then v is eligible for a *MatchFirst* and that it will write *True* in its end_v -variable while all properties of Point 3 will still hold in C_3 . Thus Point 5 holds. \square

Theorem 2. *In any execution, a matched node u can write $end_u := True$ at most twice.*

Proof. Let (u, v) be a matched edge and \mathcal{E} be an execution where u writes *True* in its variable end_u at least twice. Let $C_0 \mapsto C_1$ be the transition where u writes *True* in end_u for the second time in \mathcal{E} . If u is First (resp. Second) in C_0 then from Lemmas 11 and 13, (resp. 14), from C_1 , neither u nor v will ever be eligible for any rule. \square

5.2 The number of times single nodes can change their *end*-variable

Recall that μ is the number of matched nodes and σ is the number of single nodes.

Lemma 15. *Let x be a single node. If x writes *True* in some transition $C_0 \mapsto C_1$ then, in C_0 , $\exists u \in matched(N(x)) : p_x = u \wedge p_u = x \wedge end_x = False \wedge end_u = True$.*

Proof. To write *True* in its *end* variable, a single node must apply *UpdateEnd*. Observe now that to apply this rule, the conditions described in the Lemma must hold. \square

Lemma 16. *Let u be a matched node. Consider an execution \mathcal{E} starting after u executed some rule and in which end_u is always *True*, except for the last configuration D of \mathcal{E} in which it may be *False*. Let $\mathcal{E} \setminus D$ be all configurations of \mathcal{E} but configuration D . In $\mathcal{E} \setminus D$, the following holds:*

- $p_u \in single(N(u))$;
- p_u remains constant.

Proof. Since $end_u = True$ in $\mathcal{E} \setminus D$, the last rule executed before \mathcal{E} is necessarily a *Match* rule. So, at the beginning of \mathcal{E} , $p_u \in single(N(u))$, otherwise, u would not have executed a *Match* rule, but an *Update* instead.

We prove now that in $\mathcal{E} \setminus D$, p_u remains constant. Assume by contradiction that there exists a transition in which p_u is modified. Let $C_0 \mapsto C_1$ be the first such transition. First, observe that in $\mathcal{E} \setminus D$, u cannot execute *ResetMatch* nor *Update* since that would set end_u to *False*. Thus u must execute a *Match* rule in $C_0 \mapsto C_1$. Since the value of p_u changes in this transition, this implies that $Ask(u) \neq p_u$ in C_0 . Thus, whatever the *Match* rule, observe now that in C_1 , end_u must be *False*, which gives a contradiction and concludes the proof. \square

Definition 3. *Let u be a matched node. We say that a transition $C_0 \mapsto C_1$ is of type "a single copies *True* from u " if it exists a single node x such that $(p_x = u \wedge p_u = x \wedge end_x = False)$ in C_0 and $end_x = True$ in C_1 . Notice that by Lemma 15, $end_u = True$ in C_0 and $x \in single(N(u))$.*

*If a transition $C_0 \mapsto C_1$ is of type "a single node copies *True* from u " and if x is the single node with $(p_x = u \wedge p_u = x \wedge end_x = False)$ in C_0 and $end_x = True$ in C_1 , then we will say x copies *True* from u .*

Lemma 17. *Let u be a matched node and \mathcal{E} be an execution. In \mathcal{E} , there are at most three transitions of type "a single copies *True* from u ".*

Proof. Let \mathcal{E} be an execution. We consider some sub-executions of \mathcal{E} .

Let \mathcal{E}_{init} be a sub-execution of \mathcal{E} that starts in the initial configuration of \mathcal{E} and that ends just after the first move of u . Let $C_0 \mapsto C_1$ be the last transition of \mathcal{E}_{init} . Observe that u does not execute any move until configuration C_0 and executes its first move in transition $C_0 \mapsto C_1$. We will write $\mathcal{E}_{init} \setminus C_1$ to denote all configurations of \mathcal{E}_{init} but the configuration C_1 . We prove that there is at most one transition of type "a single copies *True* from u " in \mathcal{E}_{init} .

There are two possible cases regarding end_u in all configuration of $\mathcal{E}_{init} \setminus C_1$: either end_u is always *True* or end_u is always *False*. If $end_u = False$ then by Definition 3, no single node can copy *True* from u in \mathcal{E}_{init} , not even in transition $C_0 \mapsto C_1$, since no single node is eligible for such a copy in C_0 . If $end_u = True$, once again, there are two cases: either (i) $(p_u = null \vee p_u \notin single(N(u)))$ in all configuration of $\mathcal{E}_{init} \setminus C_1$, or (ii) $(p_u \in single(N(u)))$ in $\mathcal{E}_{init} \setminus C_1$. In case (i) then by Definition 3 no single node can copy *True* from u in \mathcal{E}_{init} , not even in $C_0 \mapsto C_1$. In case (ii), observe that p_u remains constant in all configurations of $\mathcal{E}_{init} \setminus C_1$, thus at most one single node can copy *True* from u in \mathcal{E}_{init} .

Let \mathcal{E}_{true} be a sub-execution of \mathcal{E} starting after u executed some rule and such that: for all configurations in \mathcal{E}_{true} but the last one, $end_u = True$. There is no constraint on the value of end_u in the last configuration of \mathcal{E}_{true} . According to Lemma 16, $p_u \in single(N(u))$ and p_u remains constant in all configurations of \mathcal{E}_{true} but the last one. This implies that at most one single can copy *True* from u in \mathcal{E}_{true} .

Let \mathcal{E}_{false} be an execution starting after u executed some rule and such that: for all configurations in \mathcal{E}_{false} but the last one, $end_u = False$. There is no constraint on the value of end_u in the last configuration of \mathcal{E}_{false} . By Definition 3, no single node will be able to copy *True* from u in \mathcal{E}_{false} .

To conclude, by Corollary 2, u can write *True* in its *end* variable at most twice. Thus, for all executions \mathcal{E} , \mathcal{E} contains exactly one sub-execution of type \mathcal{E}_{init} , and at most two sub-executions of type \mathcal{E}_{true} and the remaining sub-executions are of type \mathcal{E}_{false} . This implies that in total, we have at most three transitions of type "*a single copies True from u*" in \mathcal{E} . \square

Lemma 18. *In any execution, the number of transitions where a single node writes True in its end variable is at most 3μ .*

Proof. Let \mathcal{E} be an execution and x be a single node. If x writes *True* in end_x in some transition of \mathcal{E} , then x necessarily executes an *UpdateEnd* rule and by Definition 3, this means x copies *True* from some matched node in this transition. Now the lemma holds by Lemma 17. \square

Lemma 19. *In any execution, the number of transitions where a single node changes the value of its end variables (from True to False or from False to True) is at most $\sigma + 6\mu$ times.*

Proof. A single node can write *True* in its *end* variable at most 3μ times, by Corollary 18. Each of this writing allows one writing from *True* to *False*, which leads to 6μ possible modifications of the *end* variables. Now, let us consider a single node x . If $end_x = False$ initially, then no more change is possible, however if $end_x = True$ initially, then one more modification from *True* to *False* is possible. Each single node can do at most one modification due to this initialization and thus the Lemma holds. \square

5.3 How many *Update* in an execution?

Definition 4. *Let u be a matched node and C be a configuration. We define $Cand(u, C) = \{x \in single(N(u)) : (p_x = u \vee end_x = False)\}$ which is the set of vertices considered by the function $BestRematch(u)$ in configuration C .*

Lemma 20. *Let u be a matched node that has already executed some rule. If there exists a transition $C_0 \mapsto C_1$ such that u is eligible for *Update* in C_1 and not in C_0 , then there exists a single node x such that $x \in Cand(u, C_0) \setminus Cand(u, C_1)$ or $x \in Cand(u, C_1) \setminus Cand(u, C_0)$. Moreover, in transition $C_0 \mapsto C_1$, x flips the value of its end variable.*

Proof. Since u has already executed some rule, to become eligible for *Update* in transition $C_0 \mapsto C_1$, necessarily the second disjunction in the *Update* rule must hold, by Lemma 10. This implies that $(\alpha_v, \beta_v) \neq BestRematch(v)$ must become *True* in $C_0 \mapsto C_1$. Now either $Lowest(Cand(u, C_0)) \notin Cand(u, C_1)$ or $\exists x \notin Cand(u, C_0)$ such that $x = Lowest(Cand(u, C_1))$. This proves the first point.

For the second point we first consider the case $x \in Cand(u, C_1)$ and $x \notin Cand(u, C_0)$. Necessarily $end_x = True \wedge p_x \neq u$ in C_0 and $end_x = False \vee p_x = u$ in C_1 . If $p_x = u$ in C_1 then in transition $C_0 \mapsto C_1$, x has executed an *UpdateP* and the second point holds. Assume now that $p_x \neq u$ in C_1 . Necessarily $end_u = False$ in C_1 and the Lemma holds.

We consider the second case in which $x \notin Cand(u, C_1)$ and $x \in Cand(u, C_0)$. Necessarily in C_1 , $p_x \neq v$ and $end_x = True$. Thus if $end_x = False$ in C_0 the lemma holds. Assume by contradiction that $end_x = True$ in C_0 . This implies $p_x = u$ in C_0 . But since in C_1 $p_x \neq u$ then x executed either *UpdateP* or *UpdateEnd* in $C_0 \mapsto C_1$ which implies $end_x = False$ in C_1 , a contradiction. This completes the proof. \square

Corollary 3. *Matched nodes can execute at most $\Delta(\sigma + 6\mu) + \mu$ times the *Update* rule.*

Proof. Initially each matched node can be eligible for an *Update*. Now, let us consider only matched nodes that have already executed a move. For such a node to become eligible for an *Update* rule, at least one single node must change the value of its *end* variable by Lemma 20. Thus, each change of the *end*

value of a single node can generate at most Δ matched nodes to be eligible for an *Update*. By Lemma 19, the number of transitions where a single node changes the value of its *end* variables is at most $\sigma + 6\mu$ times. Thus we obtain at most $\Delta(\sigma + 6\mu)$ *Update* generated by a change of the *end* value of a single node and the Lemma holds. \square

5.4 A bound on the total number of moves in any execution

Definition 5. In the following, we call \mathcal{F} , a finite execution that does not contain any executed *Update* rule. Let $D_{\mathcal{E}}$ be the first configuration of \mathcal{F} and $D'_{\mathcal{E}}$ be the last one.

Observe that in the execution \mathcal{F} , all variables α and β remain constant and thus, predicates *AskFirst* and *AskSecond* for all matched nodes remains constant too.

Lemma 21. Let (u, v) be a matched edge. If $\text{Ask}(u) = \text{Ask}(v) = \text{null}$ in \mathcal{F} , then u and v can both execute at most one *ResetMatch*.

Proof. Recall that in the execution \mathcal{F} , by definition, u and v do not execute the *Update* rule. Moreover, these two nodes are not eligible for *Match* rules since $\text{Ask}(u) = \text{Ask}(v) = \text{null}$. Thus they are only eligible for *ResetMatch*. Observe now it is not possible to execute this rule twice in a row, which completes the proof. \square

Lemma 22. Let (u, v) be a matched edge. Assume that in \mathcal{F} , u is *First* and v is *Second*. If s_u is *False* in all configurations of \mathcal{F} but the last one, then v can execute at most one rule in \mathcal{F} .

Proof. Since $s_u = \text{False}$ in all configurations of \mathcal{F} but the last one, node v which is *Second* can only be eligible for *ResetMatch*. Observe that if v executes *ResetMatch*, it is not eligible for a rule anymore and the Lemma holds. \square

Lemma 23. Let (u, v) be a matched edge. Assume that in \mathcal{F} , u is *First* and v is *Second*. If s_u is *False* throughout \mathcal{F} , then u can execute at most one rule in \mathcal{F} .

Proof. Node u can only be eligible for *MatchFirst*. Assume u executes *MatchFirst* for the first time in some transition $C_0 \mapsto C_1$, then in C_1 , necessarily, $p_u = \text{AskFirst}(u)$, $s_u = \text{False}$ (by hypothesis) and $\text{end}_u = \text{False}$ by Lemma 8. Let \mathcal{F}_1 be the execution starting in C_1 and finishing in $D'_{\mathcal{E}}$. Since in \mathcal{F}_1 , there is no *Update*, observe that $p_u = \text{AskFirst}(u)$ remains *True* in this execution. Assume by contradiction that u executes another *MatchFirst* in \mathcal{F}_1 . Consider the first transition $C_2 \mapsto C_3$ after C_1 when it executes this rule. Notice that between C_1 and C_2 it does not execute rules. Thus in C_2 , $p_u = \text{AskFirst}(u)$, $s_u = \text{False}$ and $\text{end}_u = \text{False}$ hold. Now if u executes *MatchFirst* in C_2 it is necessarily to modify the value of s_u or end_u . By definition, it cannot change the value of s_u . Moreover it cannot modify the value of end_u as this would imply by Lemma 8 that $s_u = \text{True}$ in C_3 . This completes the proof. \square

Lemma 24. Let (u, v) be a matched edge. Assume that in \mathcal{F} , u is *First*, v is *Second* and that u writes *True* in s_u in some transition of \mathcal{F} . Let $C_0 \mapsto C_1$ be the transition in \mathcal{F} in which u writes *True* in s_u for the first time. Let \mathcal{F}_1 be the execution starting in C_1 and finishing in $D'_{\mathcal{E}}$. In \mathcal{F}_1 , u can apply at most 3 rules and v at most 2.

Proof. We first prove that in \mathcal{F}_1 , s_u remains *True*. Observe that u cannot execute *Update* neither *ResetMatch* since it is *First*. So u can only execute *MatchFirst* in \mathcal{F}_1 . For u to write *False* in s_u , it must exist a configuration in \mathcal{F}_1 such that $p_u \neq \text{AskFirst}(u) \vee p_{p_u} \neq u \vee p_v \notin \{\text{AskSecond}(v), \text{null}\}$. Let us prove that none of these cases are possible.

Since u executed *MatchFirst* in transition $C_0 \mapsto C_1$ writing *True* in s_u then, by definition of this rule, $p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge p_v \in \{\text{AskSecond}(v), \text{null}\}$ holds in C_0 . As there is no *Update* in \mathcal{F} , then $\text{AskFirst}(u)$ and $\text{AskSecond}(v)$ remains constant throughout \mathcal{F} (and \mathcal{F}_1). So each time u executes a *MatchFirst*, it writes the same value $\text{AskFirst}(u)$ in its p value. Thus $p_u = \text{AskFirst}(u)$ holds throughout \mathcal{F}_1 . Moreover, each time v executes a rule, it writes either *null* or the same value $\text{AskSecond}(v)$ in its p value. Thus $p_v \in \{\text{AskSecond}(v), \text{null}\}$ holds throughout \mathcal{F}_1 . Now by Lemma 9, in C_1 we have, $\exists x \in \text{single}(N(u)) : p_u = x \wedge p_x = u$, since $s_u = \text{True}$. This stays *True* in \mathcal{F}_1 as p_u remains constant and x will then not be eligible for *UpdateP* in \mathcal{F}_1 . Thus $p_{p_u} = u$ holds throughout \mathcal{F}_1 .

Thus, $p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge p_v \in \{\text{AskSecond}(v), \text{null}\}$ holds throughout \mathcal{F}_1 and so $s_u = \text{True}$ throughout \mathcal{F}_1 .

This implies that in \mathcal{F}_1 , v is only eligible for *MatchSecond*. The first time it executes this rule in some transition $B_0 \mapsto B_1$, with $B_1 \geq C_1$, then in B_1 , $p_v = \text{AskSecond}(v)$, $s_v = \text{end}_v$ and this will hold between B_1 and $D'_\mathcal{E}$. If $\text{end}_v = \text{True}$ in B_1 then this will stay *True* between B_1 and $D'_\mathcal{E}$. Indeed, p_v is not eligible for *UpdateP* and we already showed that $p_u = \text{AskFirst}(u)$ holds in \mathcal{F}_1 . In that case, between B_1 and $D'_\mathcal{E}$, v will not be eligible for any rule and so v will have executed at most one rule in \mathcal{F}_1 . In the other case, that is $\text{end}_v (= s_v) = \text{False}$ in B_1 , since $p_v = \text{AskSecond}(v)$ holds between B_1 and $D'_\mathcal{E}$, necessarily, the next time v executes a *MatchSecond* rule, it is to write *True* in end_v . After that observe that v is not eligible for any rule. Thus, v can execute at most 2 rules in \mathcal{F}_1 .

To conclude the proof it remains to count the number of moves of u in \mathcal{F}_1 . Recall that we proved that s_u is always *True* in \mathcal{F}_1 . Thus whenever u executes a *MatchFirst*, it is to modify the value of its *end* variable. Observe that this value depends in fact of the value of end_v and of p_v since we proved $p_u = \text{AskFirst}(u) \wedge p_{p_u} = u \wedge s_u \wedge p_v \in \{\text{AskSecond}(v), \text{null}\}$ holds throughout \mathcal{F}_1 . Since we proved that in \mathcal{F}_1 , v can execute at most two rules, this implies that these variables can have at most three different values in \mathcal{F}_1 . Thus u can execute at most 3 rules in \mathcal{F}_1 . \square

Lemma 25. *Let (u, v) be a matched edge. Assume that in \mathcal{F} , u is First and v is Second. If s_u is True throughout \mathcal{F} and if u does not execute any move in \mathcal{F} , then v can execute at most two rules in \mathcal{F} .*

Proof. By Definition 5, v cannot execute *Update* in \mathcal{F}_1 . Since we suppose that in \mathcal{F}_1 , $s_u = \text{True}$ then v is not eligible for *ResetMatch*. Thus in \mathcal{F}_1 , v can only execute *MatchSecond*. After it executed this rule for the first time, $p_v = \text{AskSecond}(v)$ and $s_v = \text{end}_v$ will always hold, since v is only eligible for *MatchSecond*. Thus the second time it executes this rule, it is necessarily to modify its end_v and s_v variables. Observe that after that, since u does not execute rules, v is not eligible for any rule. \square

Lemma 26. *Let (u, v) be a matched edge. In \mathcal{F} , u and v can globally execute at most 12 rules.*

Proof. If $\text{Ask}(u) = \text{Ask}(v) = \text{null}$, the Lemma holds by Lemma 21. Assume now that u is *First* and v is *Second*. We consider two executions in \mathcal{F} .

Let $C_0 \mapsto C_1$ be the first transition in \mathcal{F} in which u executes a rule. Let \mathcal{F}_0 be the execution starting in $D_\mathcal{E}$ and finishing in C_0 . There are two cases.

If $s_u = \text{False}$ in \mathcal{F}_0 then v is only eligible for *ResetMatch* in this execution. Observe that after it executes this rule for the first time in \mathcal{F}_0 , it is not eligible for any rule after that in \mathcal{F}_0 .

If $s_u = \text{True}$ in \mathcal{F}_0 then by Lemma 25, v can execute at most two rules in this execution. In transition $C_0 \mapsto C_1$, u and v can execute one rule each.

Let \mathcal{F}_1 be the execution starting in C_1 and finishing in $D'_\mathcal{E}$. Whatever rule u executes in transition $C_0 \mapsto C_1$ observe that u either writes *True* or *False* in s_u . If u writes *True* in s_u in transition $C_0 \mapsto C_1$, then by Lemma 24, u and v can execute at most five rules in total in \mathcal{F}_1 .

Consider the other case in which u writes *False* in C_1 . Let $C_2 \mapsto C_3$ be the first transition in \mathcal{F}_1 in which u writes *True* in s_u . Call \mathcal{F}_{10} the execution between C_1 and C_3 and \mathcal{F}_{11} the execution between C_3 and $D'_\mathcal{E}$. By definition, s_u stays *False* in $\mathcal{F}_{10} \setminus C_3$. Thus in $\mathcal{F}_{10} \setminus C_3$, u can execute at most one rule, by Lemma 23. Now in \mathcal{F}_{10} , u can execute at most two rules. By Lemma 22, v can execute at most one rule in \mathcal{F}_{10} . In total, u and v can execute at most three rules in \mathcal{F}_{10} . In \mathcal{F}_{11} , u and v can execute at most five rules by Lemma 24. Thus in \mathcal{F}_1 , u and v can apply at most eight rules. \square

Theorem 3. *In any execution, matched nodes can execute at most $12\mu(\Delta(\sigma + 6\mu) + \mu)$ rules.*

Proof. By Lemma 3, matched nodes can execute at most $\Delta(\sigma + 6\mu) + \mu$ times the *Update* rule. By Lemma 26, between the execution of two *Updates* rules, a matched node can execute at most 12 rules, which concludes the proof. \square

Lemma 27. *In any execution, single nodes can execute at most σ times the *ResetEnd* rule.*

Proof. We prove that a single node x can execute the *ResetEnd* rule at most once. Assume by contradiction that it executes this rule twice. Let $C_0 \mapsto C_1$ be the transition when it executes it the second time. In C_0 , $\text{end}_x = \text{True}$, by definition of the rule. Since x already executed a *ResetEnd* rule, it must have some point wrote *True* in end_x . This is only possible through an execution of *UpdateEnd*. Thus

consider the last transition $D_0 \mapsto D_1$ in which it executed this rule. Observe that $D_1 \leq C_0$. Since between D_1 and C_0 , end_x remains *True*, observe that x does not execute any rule between these two configurations. Now since in D_1 , $p_x \neq null$ and this holds in C_0 then x is not eligible for *ResetEnd* in C_0 , which gives the contradiction. This implies that single nodes can execute at most $\mathcal{O}(\sigma)$ times the *ResetEnd* rule. \square

Lemma 28. *In any execution, single nodes can execute at most $\sigma + 6\mu$ times the *UpdateEnd* rule.*

Proof. By Lemma 19, single nodes can change the value of their *end* variable at most $\sigma + 6\mu$ times. Thus they can apply *UpdateEnd* at most $\sigma + 6\mu$ times, since in every application of this rule, the value of the *end* variable must change. \square

Lemma 29. *In any execution, single nodes can execute at most $12\mu \times (\Delta(\sigma + 6\mu) + \mu) + 1$ times the *UpdateP* rule.*

Proof. Let x be a single node. Let $C_0 \mapsto C_1$ be a transition in which x executes an *UpdateP* rule and let $C_2 \mapsto C_3$ be the next transition after C_1 in which x executes an *UpdateP* rule. We prove that for x to execute the *UpdateP* rule in $C_2 \mapsto C_3$, a matched node had to execute a move between C_0 and C_2 .

In C_1 there are two cases: either $p_x = null$ or $p_x \neq null$. Assume to begin that $p_x = null$. This implies that in C_0 the set $\{w \in N(x) | p_w = x\}$ is empty. In C_2 , $p_x = null$, since between C_1 and C_2 , x can only apply *UpdateEnd* or *ResetEnd*. Thus if it applies *UpdateP* in C_2 , necessarily $\{w \in N(x) | p_w = x\} \neq \emptyset$. This implies that a matched node must have executed a *Match* rule between C_1 and C_2 and the lemma holds in that case.

Consider now the case in which $p_x = u$ with $u \neq null$ in C_1 . By definition of the *UpdateP* rule, we also have $u \in matched(N(x)) \wedge p_u = x$ holds in C_0 . In C_2 we still have that $p_x = u$ since between C_1 and C_2 , x can only execute *UpdateEnd* or *ResetEnd*. Thus if x executes *UpdateP* in C_2 , necessarily $p_{p_x} \neq x$. This implies that $p_u \neq x$ and so u executed a rule between C_0 and C_2 .

Now, the lemma holds by Theorem 3. \square

Corollary 4. *In any execution, nodes can execute at most $\mathcal{O}(n^3)$ moves.*

Proof. According to Lemmas 27, 28 and 29, single nodes can execute at most $\mathcal{O}(n^3)$ moves. Moreover, according to Theorem 3, matched nodes can execute at most $\mathcal{O}(n^3)$ moves. \square

Corollary 5. *The algorithms MAXMATCH converges in $\mathcal{O}(n^3)$ steps under the adversarial distributed daemon.*

References

- [1] Johanne Cohen, Khaled Maâmra, George Manoussakis, and Laurence Pilard. The Manne *et al.* self-stabilizing 2/3-approximation matching algorithm is sub-exponential. *ArXiv e-prints*, April 2016. [arXiv:1604.08066](#).
- [2] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.
- [3] Doratha E. Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Inf. Process. Lett.*, 85(4):211–213, 2003.
- [4] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.
- [5] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A new self-stabilizing maximal matching algorithm. *Theoretical Computer Science (TCS)*, 410(14):1336–1345, 2009.
- [6] Fredrik Manne, Morten Mjelde, Laurence Pilard, and Sébastien Tixeuil. A self-stabilizing 2/3-approximation algorithm for the maximum matching problem. *Theoretical Computer Science (TCS)*, 412(40):5515–5526, 2011.
- [7] Robert Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *16th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, Lecture Notes in Computer Science, pages 259–269. Springer, 1999.